



Accompanying notes to the presentation and session

Compiled by: Bertie Buitendag



Contents

1. Introduction to OOP in Delphi	2
1.1 Object oriented programming	3
1.1.1 Introduction	3
1.1.2 What is OOP?	3
1.1.3 OOP in terms of the concepts involved	5
1.1.4 Working with a Delphi GUI Application	12
1.1.5 Creating and implementing a user defined class in a Delphi application	18
1.1.6 Implementing the class as part of an application.	22
1.1.7 Discussion pertaining to the TMyTime implementation application	23
1.1.8 Sample runs of the application	26



Disclaimer: This set of notes is presented under ULA terms (Use it, Lose it, or Abuse it) conditions and is presented as is. 😊

1. Introduction to OOP in Delphi

Content

Basic concepts

- Abstraction

- Classes vs Objects

- Creating a user defined class in Delphi

- Using a user defined class in an application

1.1 Object oriented programming

1.1.1 Introduction

OOP (Object Oriented Programming) is a programming paradigm that was introduced to the IT world in the late 1960's. It is said that the first OO language was SIMULA I and SIMULA 67. Other pioneering OOP languages were: SmallTalk (1970's), and C++ (1980's) of which the latter is still very popular today. In contrast to procedural programming which was and still is very popular today OOP requires a total new way of thinking especially for those who was and are familiar with procedural programming.

Within the OOP milieu one would often hear various terms. The next figure lists some of these concepts and terms.



Figure 1.1 OOP concepts and terms



OOP is a programming methodology which incorporates all the traditional features of procedural programs but with several enhancements. OOP however requires a different way of thinking and adds several new concepts to programming. (Farrel, 2002:10)

OOP allows programmers to reuse and modify existing objects resulting in faster program creation. OOP languages are normally event driven, where events are actions to which a program responds, such as pressing a key on a keyboard, typing a value in a text box etc. (Shelly, et al. 2005, 1.17)

1.1.2 What is OOP?

OO or Object Orientation tries to model real world objects (in a software format) in terms of its actions and its attributes.

Most visual software applications aim to model processes on real world objects e.g., performing a sale of Items at a Point-of-Sale terminal or making a booking for a concert etc.

The control panel of your operating system is nothing more than an object that was modelled by software in order to help you control and manage computer. E.g., the Date and Time interface provided by the operating system allows a user to interact with the computers, date and time. Not only does the interface allow you to set the date and time it also allows the user to read and obtain the current date and time and set other interface values and properties such as the region etc.

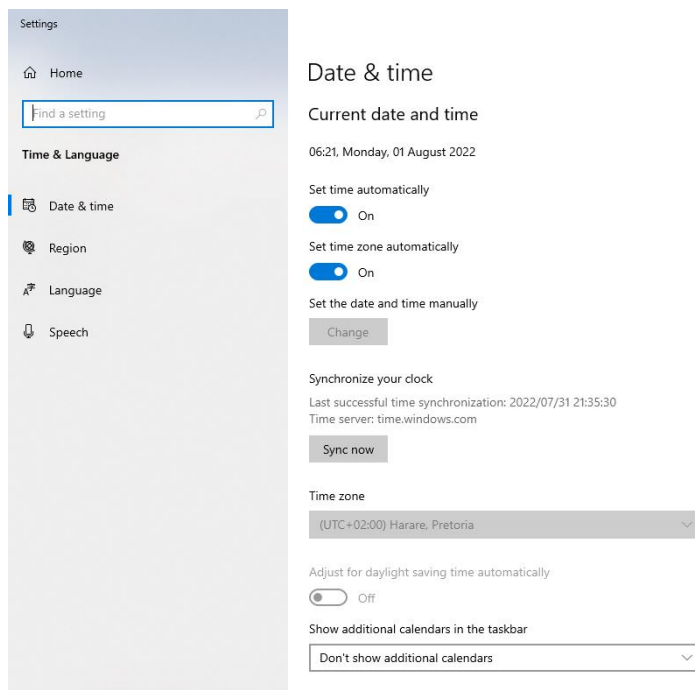


Figure 1.2 Date and time settings in the Operating System

Wikipedia (2008) defines OOP by stating that: Object-oriented programming (OOP) is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. Programming techniques may include features such as information hiding, data abstraction, encapsulation, modularity, polymorphism, and inheritance. It was not commonly used in mainstream software application development until the early 1990s. Many modern programming languages now support OOP.

In other words, OOP aims to create software that models or simulate real world objects. OOP aims to simulate the object in terms of its attributes (the characteristics of the object) and its behaviour (what the object can do).

Many programming languages which support EDP (Event Driven Programming) are based on OO. Users create computer programs which models real world scenarios. The program interface, normally a GUI is composed of various components which reacts to user events such as clicking on a button opening a form etc.

In the screenshot above which models a watch and a calendar (both objects) via a GUI the user is allowed to either obtain (read the time) or set a new time, the same applies to the date. The watch object models the time for the user while the calendar models the date.

1.1.3 OOP in terms of the concepts involved

Figure 1.1 depicted some of the concepts normally associated with OOP the following section would provide an overview of each of the concepts.

ADT (Abstract Data Type)

Delphi (Object Pascal) has a lot of common built in data types which allows a user to easily define a variable to represent and store a value for a certain item or descriptor, e.g., Integer to declare variables to store whole numbers, Real for floating point values and Boolean for flags.

Sometimes however more than one simple descriptor is needed to define/declare an object. A concept such as a rectangle needs to be defined in terms of its length and its width both of which could in turn be stored as decimals.

So Simple value variables are normally described by a single descriptor:

e.g. Length
 Colour
 Distance
 Name

Complex objects are described by using more than one simple attribute.

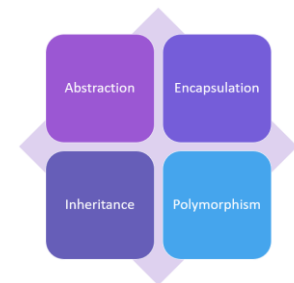
e.g. Motorcar
 Engine
 Doors
 Make
 Model
 Year
 Transmission type
 etc.

A Motorcar is an abstract concept since more than one simple descriptor is needed to describe the car.



In order for the programming language to be considered a true OO programming language it must support the following OO features.

In this sidenote section an overview of each of the concepts are provided which is covered at a later stage as well.



Abstraction

The first of these features that characterizes OOP languages is the idea of abstraction. Abstraction in this context signifies the ability of the language to model the real world characteristics of the problem the program is trying to solve. For example, if you are creating a program that handles customer orders, it is much easier to deal with a language where you can create something called a customer (which has the important characteristics of a customer) than to deal with data structures that do not have any inherent meaning. Abstraction refers to the language's ability to model the real world. That is exactly what we are trying to do in the creation of computer programs—model on the computer something that is happening in the physical world.

ADT (Abstract Data type) can therefore be seen as: a user- defined data type consisting of (being described by) several simple data types.

E.g. Brick (length, width, height)
 Date (year, month, day)
 Time (hours, minutes, seconds)

An Abstract Data Type is used to declare objects that cannot be defined by using concrete / simple data types.

Classes

Classes are the design plans for building and describing an object. A class describes an object in terms of what the object can do (its behaviour) as well as what the object is (its attributes).

A Building plan is an example of a class, with one building plan a builder will be able to build and construct many buildings, all with the same attributes.

Many textbooks state that classes are the blueprints for defining objects, in other words a class allows someone or something to construct an object. Robots are programmed with "classes" to construct cars, while a watch factory's robots construct watches from a class. A carpenter may build a wooden box from a plan which could be seen as a class.

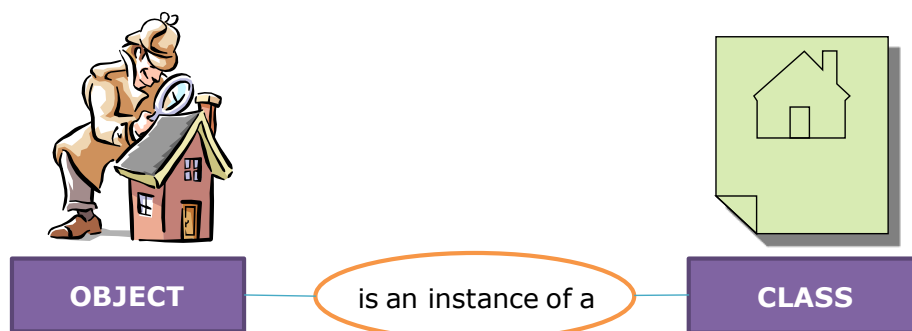


Figure 1.3 Object is an instance of a class

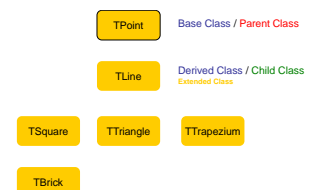
The design specification as set aside in the class ensures that the object created conforms to the plan. E.g., if a rectangle class are to describe a rectangle in terms of its length and its width, the rectangle class would ensure that every subsequent rectangle has a length and a width.

Encapsulation

Encapsulation requires that every program module be as self-contained as possible, and it should not have any side effects that affect other modules. The goal here is to create modules (objects) that can be used whenever they are appropriate, without having to worry about special requirements or side effects. For example, you do not want your dishwasher to start running every time you turn on the VCR.



Inheritance



Inheritance refers to the ability for one object to inherit the general characteristics of another object, and then add specific capabilities. It allows you to reuse common code, so it cuts down on the amount of code that must be written.

The reason that this capability is so important in a programming language is that the real world is made up of objects that fit into a hierarchy. The more you think about this, the more you realize that the real world is full of objects, and an object-oriented programming language means that we can more closely model the real world's objects.



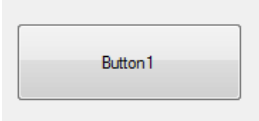
Polymorphism

The last requirement for an object-oriented language is polymorphism. Polymorphism is the ability for two related but different objects to interpret the same command, but act on it in a way that is specific to them. For example, you can give the command 'to walk' to both man and dog, and each of them will know how to walk. However, they go about fulfilling

Objects

Objects are real world things which have been created at some or other stage; objects may be tangible or non-tangible. Most real-world objects are tangible while software objects are predominantly non-tangible.

this command, differently—the dog walks on all fours while the man walks upright.

Watch	Car	Button
A Watch is an example of an object; the watch has various attributes; colour strap-size, shape, time, buttons etc.	A Car is an example of an object it is described in terms of its various attributes (the things that make the car what it is) and its behaviour (what the car can do).	A Button is an example of a software object; it is described in terms of its various properties and behaviour.
		



Fantastic introductory video on OOP made by Gerhard Visagie



<https://www.youtube.com/watch?v=fDxju-4vSto&t=2s>

Objects are normally described in terms of attributes and behaviour.

The **attributes** of an object describe what the object is in the example of the watch a watch could be described in terms of:

- 🏠 FaceShape
- 🏠 Colour
- 🏠 Straptype
- 🏠 NrOfButtons
- 🏠 WatchShape
- 🏠 Time
- 🏠 DigitalOrAnalog
- 🏠 Etc

The behaviour of the watch describes what the watch can do:

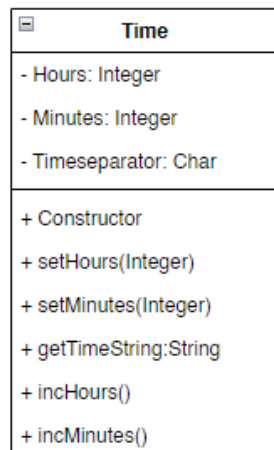
- 🏠 Set the Hours
- 🏠 Set the Minutes
- 🏠 Read the Time
- 🏠 Set the Alarm
- 🏠 Sound the Alarm etc

The behaviour of the watch describes the functionality of the object.



UML Class Diagrams

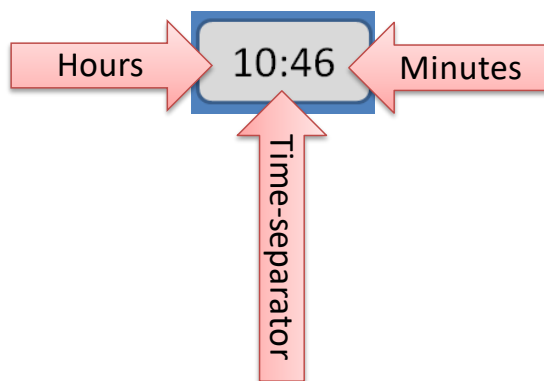
The Unified Modelling Language uses class diagrams to allow programmers to model classes diagrammatically.

The following UML class diagram represents a Time class:



The Time class has 3 attributes as indicated by the diagram. The attributes are:

-  Hours
-  Minutes
-  TimeSeperator



Each of these attributes are represented and stored by using a concrete data type

- Hours is stored as an *Integer*
- Minutes is stored as an *Integer*
- TimeSeperator is stored as a *Char*

These three attributes are used to represent the “data” or the “fields” of the time object.

TAKE NOTE: The UML class diagram only represents the TIME aspect (object) that is part of the watch.

Within the watch the actual time is hidden away from the user, and the time is only displayed as a set of characters (a string) on the screen of the digital watch. The user cannot “touch” the actual time inside of the watch, but the watch allows the user to change the time by using buttons on the watch.

Objects normally have an interface that allows a user to communicate with the values and the attributes of the object. The interface of the watch allows the user to:

- Set the hours
- Set the minutes
- Increment the hours with 1
- Increment the minutes with 1
- Read the time as a string

Objects are commonly described in two parts the visible interface part which the user is allowed to interact with and the non-visible hidden part. The visible parts normally interact with the non-visible parts.

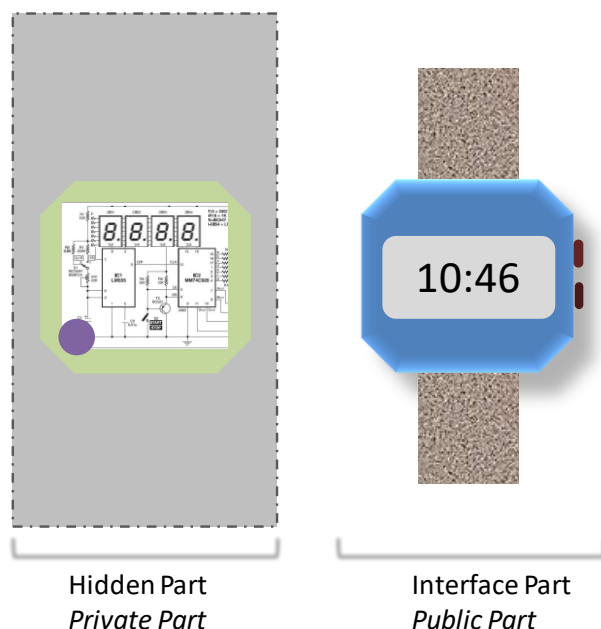



Figure 1.4 Analogy of Private and Public access specifiers



The attributes indicated with a preceding – in the class diagram is indicated to be private or hidden from the user. The methods or attributes preceded with a + indicates that it is visible or public to the user.

A user of an object only has access to the public methods or fields of the object.

The time class is therefore modelled in terms of its three attributes as well as its behaviour what it can do. A class diagram indicates to the programmer which of the attributes are visible to the user and which of the attributes are hidden away.

Most methods of an object are normally accessible to the user of the objects. Methods are functions and subroutines used to code some functionality pertaining to the object.

The concept of hiding some of the functionality and attributes of an object is called encapsulation. A watch is encapsulated in order to hide and protect its inner parts.

Attributes

We have seen that attributes are the descriptors of a class which is to represent an object. Attributes are normally defined using built-in data types. An attribute that is used to describe a class may also be an object by itself.

E.g.: a Watch may contain a time object; the same time object may also be used in an alarm clock, as well as a microwave. The reusability of objects in programming is what makes OOP such a powerful software development concept. A programmer does not need to write the code for the Time class instead the programmer may decide to incorporate a current available class into his solution. EDP utilises this concept fully, all components placed on a form are actually created from a particular class.

Using one object as part of a new object is called composition. Another good example is that of a car a car comprises of various other objects. E.g., motor, gearbox, seats, wheels, etc.

Methods

The methods of the class represent what the class can do. Methods in program code are presented as subroutines which may be functions or procedures. There are various types of methods, some methods are used to obtain a value from a user and to set the value of an attribute. Other methods are used to retrieve the value of an attribute and to send it to the user, while other methods are used to perform certain functionality of the object.

TAKE NOTE: It is advisable at this stage to do a thorough review of subroutines, functions, and procedures.

- 🎓 Accessor methods are used to retrieve the value of an attribute. Accessor methods are sometimes preceded with the word get. E.g., **getTime**
- 🎓 Mutator methods are used to assign a value from the user to an attribute. Mutator methods are sometimes preceded with the word set. E.g., **setHours**

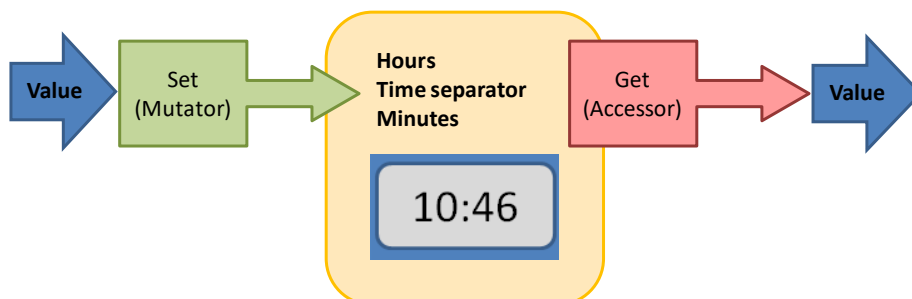



Figure 1.5 Accessor and Mutator

Constructors are methods that are used to set the default value for objects when they are created. When a bunch of watches are switched on that is of the same type and same manufacturer the watch would have been pre-programmed with a default time.

In Delphi methods are defined to classes by means of procedures and functions.

Properties

Properties are special attributes of a class that is usually only read and assigned a value.



The text/caption attribute of a Button or a Label for example in is considered a property.

The user may set the value to the property or get the value for a calculation.

Encapsulation

Encapsulation is the term that is used to describe how an object's attributes are hidden. Information hiding is one of the benefits of OOP where the internal data is hidden from the actual functioning of the object. Encapsulation hides the internal attributes of a class to protect the data and to separate the class logic. Encapsulation is the method of dividing and defining an object into a visible and non-visible part covering up the data and the attributes as well as the function behaviours into a class.

Just as a capsule hide and protects the inner parts of the capsule just so does encapsulation.

A Microwave object is encapsulated into two parts (a private inner part of which you don't want to interfere with) and an outside interface for the user to interact with. The same applies for the watch object.

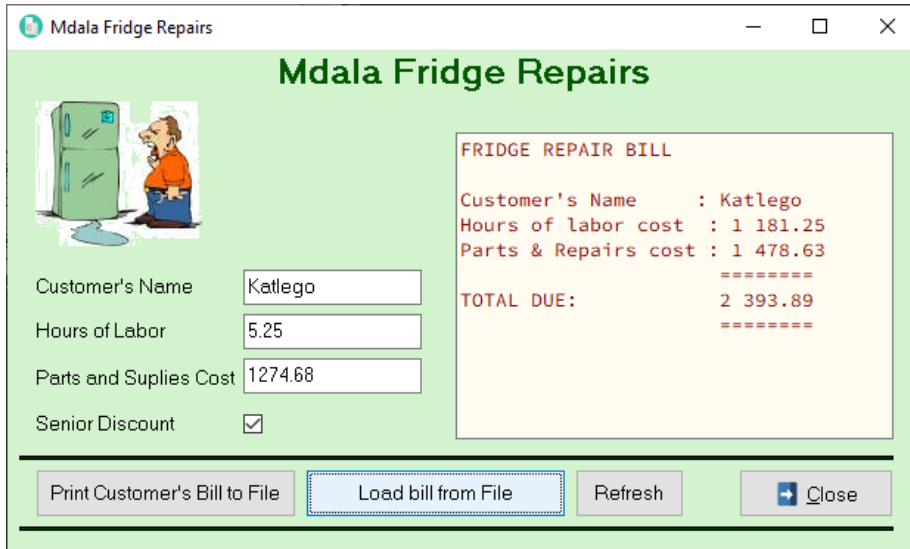
Instantiation

When an object is made it is said that the object is instantiated. We have learned that an object is an instance of a class, so making the object is instantiating it.

1.1.4 Working with a Delphi GUI Application

With all prior applications that you may have developed, you would have implemented OOP without you even knowing it. If you study the code for a simple rudimentary Delphi application, you will notice that you have added objects and used attributes in your form. You have even created event handlers and passed messages to and from created objects on your form.

Study the following GUI used as part of a simple application that calculates the cost pertaining to the repair of a fridge. An invoice is generated that is written to a text file, whereafter the user has the option to load the invoice text file for display purposes.



Code for the application

```
unit MainUnit;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, ExtCtrls;

type
  TfrmMain = class(TForm)
    lblName: TLabel;
    lblHours: TLabel;
    lblParts: TLabel;
    edtName: TEdit;
    edtHours: TEdit;
    edtParts: TEdit;
    mmoDisplay: TMemo;
    lblMainTitle: TLabel;
    btnPrintToFile: TButton;
    btnRefresh: TButton;
    bbtnClose: TBitBtn;
    Image1: TImage;
    btnLoadFromFile: TButton;
    Shape1: TShape;
    Shape2: TShape;
  end;
```

```

    lblSeniorDiscount: TLabel;
    ckbDiscount: TCheckBox;
    procedure btnPrintToFileClick(Sender: TObject);
    procedure btnRefreshClick(Sender: TObject);
    procedure btnLoadFromFileClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    frmMain: TfrmMain;
    { Declare Constants }
    const Rate: Double = 225.0;
    const Tax: Double = 0.16;

implementation

{$R *.dfm}

procedure TfrmMain.btnPrintToFileClick(Sender: TObject);
var
    { Declare Variables }
    Name: String;
    rHours, rHoursCost, rParts, rPartsCost, rTotal: Real;
    fileWriter: TStreamWriter;

begin
    { Get data from Edit Boxes
      NOTE the StrToFloat function to change strings to float numbers }
    Name := edtName.Text;
    rHours := StrToFloat(edtHours.Text);
    rParts := StrToFloat(edtParts.Text);

    { Calculate the total amount}
    rHoursCost := rHours * Rate;
    rPartsCost := rParts + (rParts * Tax);
    rTotal := rHoursCost + rPartsCost;

    if ckbDiscount.Checked then
    begin
        rTotal := rTotal * 0.9;
    end;

    { Create a file stream and open a text writer for it }
    fileWriter := TStreamWriter.Create(
    TFileStream.Create(Name + '_Invoice.txt', fmCreate),
    TEncoding.UTF8);

    { Write the data to the textfile}

```

```

fileWriter.WriteLine('FRIDGE REPAIR BILL');
fileWriter.WriteLine(' ');
fileWriter.WriteLine('Customer's Name      : ' + Name);
fileWriter.WriteLine('Hours of labor cost : '
    + FormatFloat('#,##0.00', rHoursCost));
fileWriter.WriteLine('Parts & Repairs cost : '
    + FormatFloat('#,##0.00', rPartsCost));
fileWriter.WriteLine('=====');
fileWriter.WriteLine('TOTAL DUE:      '
    + FormatFloat('#,##0.00', rTotal));
fileWriter.WriteLine('=====');

fileWriter.Flush;
fileWriter.Close;
fileWriter.BaseStream.Free;
fileWriter.Free();

btnLoadFromFile.Enabled := true;

end;

procedure TfrmMain.btnRefreshClick(Sender: TObject);
begin
    // Clear all edits and the memo
    edtHours.Text := '';
    edtName.Text := '';
    edtParts.Text := '';

    edtName.SetFocus;

    ckbDiscount.Checked := false;
    mmoDisplay.Lines.Clear();
    btnLoadFromFile.Enabled := false;
end;

procedure TfrmMain.FormShow(Sender: TObject);
begin
    btnRefreshClick(Self);
end;

procedure TfrmMain.btnLoadFromFileClick(Sender: TObject);
begin
    mmoDisplay.Clear;
    mmoDisplay.Lines.LoadFromFile(edtName.Text + '_Invoice.txt')
end;

end.

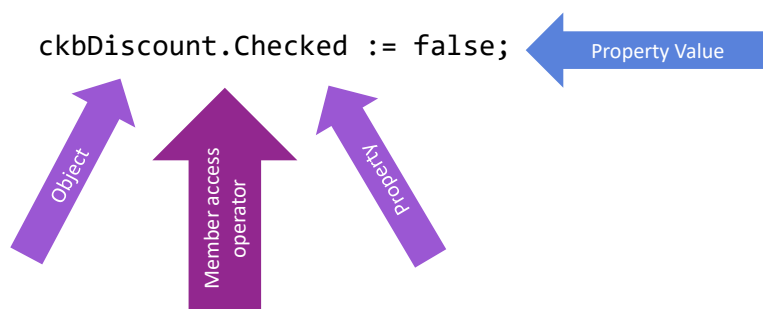
```


Notes on the program

In the application various objects are deployed and implemented. Amongst others, components such as editboxes, buttons and a checkbox. In order to create the invoice and write the data to a textfile a TStreamWriter object is also used.

The checkbox is used to indicate whether a 10% senior citizen discount is to be applied to the final amount due. In the btnRefreshClick event the following line of code is added to ensure that the checkbox is not checked initially.

Assigning a value to a property of an Object

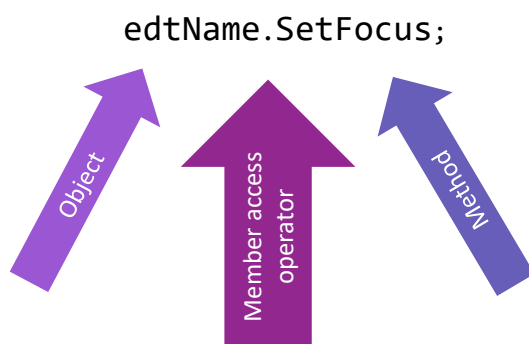


See more:

<https://docwiki.embarcadero.com/Libraries/Alexandria/en/System.Classes>

The SetFocus method of the edtName editbox places the cursor in the editbox.

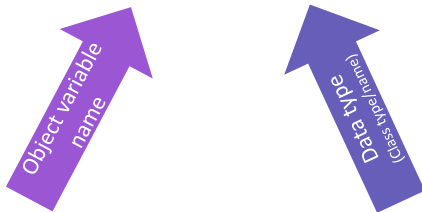
Invoking a method of an Object



In the following snippet a system class object is declared the same way that a normal variable is defined, but with a difference. The variable name `fileWriter` is set to reference a `TStreamWriter` object in memory. The `fileWriter` object however has not been created yet.

In order to do so, a special method named the constructor needs to be invoked / called.

```
procedure TfrmMain.btnPrintToFileClick(Sender: TObject);  
var  
  { Declare Variables }  
  Name: String;  
  rHours, rHoursCost, rParts, rPartsCost, rTotal: Real;  
  fileWriter: TStreamWriter;
```



As part of the `btnPrintToFileClick` on click event handler, the `fileWriter` object is instantiated by means of the invocation of the constructor method of the class called `Create`.

```
{ Create a file stream and open a text writer for it }  
fileWriter := TStreamWriter.Create(  
  TFileStream.Create(Name + '_Invoice.txt', fmCreate),  
  TEncoding.UTF8);
```

TStreamWriter object named `fileWriter` Created calling the constructor

```
{ Write the data to the textfile}
```

```
fileWriter.WriteLine('FRIDGE REPAIR BILL');  
.  
.  
.
```

```
fileWriter.Flush;  
fileWriter.Close;  
fileWriter.BaseStream.Free;  
fileWriter.Free();
```

After the object has been created various methods can be called and invoked.

1.1.5 Creating and implementing a user defined class in a Delphi application

The Delphi IDE is fully developed using OOP principles. The IDE allows a user to develop amongst others, solutions with programmer defined classes. Read through the following example and follow the steps.

Example 1 – Implementing the Time Class in a GUI Application

Develop a class named, TMyTime. The class comprises of two attributes Hours and Minutes and a property Timeseparator. The timeseparator property is used to set the character to be used for display purposes of the time, it should default to a colon ::

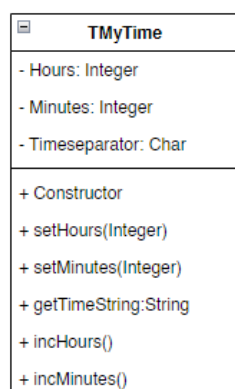
The class should include methods to allow the user to set the hours and the minutes and to retrieve the time as a string in the format HH:MM. The time should be padded with leading 0's where applicable.

Five minutes past eight should be returned as 08:05. All newly created Time objects should default to 12:00.

The class should also include a method that will increment the hours with one hour and a method that will increment the minutes with one minute. These two methods should ensure that the data manipulated stay valid. When invalid data is sent to the SetHours and SetMinutes methods respectively the data values of the corresponding attributes should not be changed.

Define your class in a separate unit. Develop an application to test your class.

Step 1: Plan your class, using a UML class diagram.



The following site has a well explained example of implementing a user defined class as part of an application.

<http://www.teachitza.com/delphi/createclass.htm>

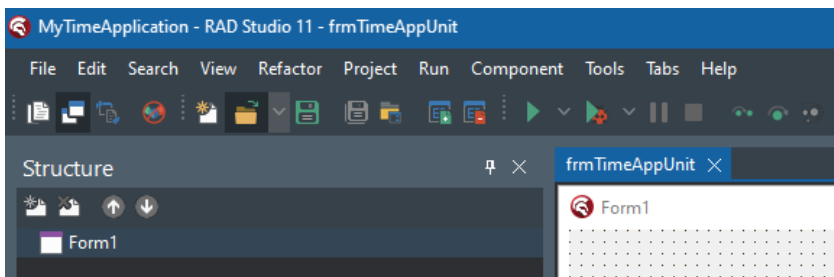
TeachITza

Step 2: Design and plan the functionality of each method

Design and plan the logic and functioning of each of the methods. A function / planning chart could be implemented at this stage. The developer needs to plan how each of the methods of the class are to be implemented with regard to the parameters and their types etc.

Step 3: Create a new VCL/FMX application and save all files appropriately.

The name of the application/project can be set to MyTimeApplication and the unit file can be saved with an appropriate name as well.



For this example, a VCL application was created, and the unit file was saved as to frmTimeAppUnit and the project as MyTimeApplication.

Step 4: Create a new VCL/FMX application and save all files appropriately.

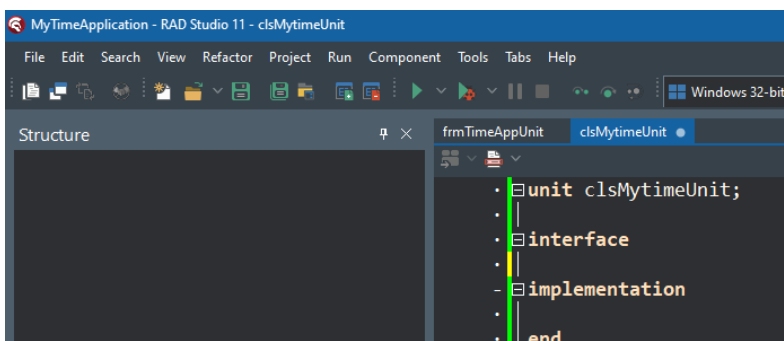
Select File->New Unit

which will create an empty unit where you can define your class.

A new unit will be created

Save the unit as *clsMyTimeUnit.pas*

An empty unit will be created.



Step 5: Define a class

Add the code for the definition of the class. The definition of a class is nothing more than the declaration of a user defined data type. This is done under the interface section of the unit.

Step 6: Add the private and public member attributes, methods, and definitions.

Using the class diagram as guide add the definitions as part of the interface section.

```
interface
Type TMyTime = class
  private
    fHours: Integer;
    fMinutes: Integer;
    fTimeSeparator : Char;
  public
    constructor create; overload;
    constructor create(HH : integer; MM : integer; TS : char);
    overload;

    procedure setHours(HH : integer);
    procedure setMinutes(MM : integer);

    function getHours:integer;
    function getMinutes:integer;

    function getSeparator:char;
    procedure setSeparator(TS : char);

    procedure incHours;
    procedure incMinutes;

    function getTimeStr:String;

    property separator : char read getSeparator write setSeparator;
end;
```

User defined ADT named **TMyTime** as a class. The **T** in front of the class name is an old Borland convention to indicate that a Type is declared.

Definition of three private field attributes of the class hidden away (encapsulated) from the user of the class. The **f** in front of the attribute names is an indicator for a field

Parameters

Overload indicates that there are two versions of a method

Mutator (**set**) methods (to mutate) or change the values of field attributes.

Accessor (**get**) methods to retrieve and return the value of field attributes.

Auxiliary methods are methods that is included that provide some or other functionality relating to the operations of the class. A **toString** method is a typical example of such a method



TERMS:

Constructor

A constructor is a special method that creates and initializes instance objects. The declaration of a constructor looks like a procedure declaration, but it begins with the word constructor

Default constructor

Normally a constructor method with no parameters which sets the attributes to some required default values.

Parameterised constructor

A constructor where some parameters are passed to be used to set the values of the class fields/attributes

Property

A property is an interface to data associated with an object (often stored in a field). Properties have access specifiers, which determine how their data is read and modified. From other parts of a program outside of the object itself, a property appears in most respects like a field.

Step 7: Add the implementation code.

The implementation section of the unit presents the developer with the area to write the implementation code (i.e. the operations and logic) for each of the methods.

Take note how the method is tied to the class name by adding the name of the class, i.e., TMyTime. as part of each method.

implementation

```
constructor TMyTime.create;
begin
    fHours := 0;
    fMinutes := 0;
    fTimeSeparator := ':';
end;
```

The default constructor assigns initial values to each of the attributes/fields

```
constructor TMyTime.create(HH : integer; MM : integer; TS : char);
begin
    fHours := HH;
    fMinutes := MM;
    fTimeSeparator := TS;
end;
```

The parameterised constructor assigns values passed via parameters to the various attributes

```
procedure TMyTime.setHours(HH : integer);
begin
    if (HH > 0) and (HH < 24) then fHours := HH;
end;
```

Mutator methods with some range checking that is applied

```
procedure TMyTime.setMinutes(MM : integer);
begin
    if (MM > 0) and (MM < 60) then fMinutes := MM;
end;
```

```
function TMyTime.getHours:integer;
begin
    result := fHours;
end;
```

Accessor methods as functions to return the applicable field/attribute value/s

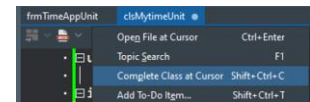
```
function TMyTime.getMinutes:integer;
begin
    result := fMinutes;
end;
```

Auxiliary method to add 1 hour to the fHours attribute with some validation applied.

```
procedure TMyTime.incHours;
begin
    if (fHours = 23) then fHours := 0 else inc(fHours);
end;
```



A Quick way to complete the method headers and shells for the defined class is to use the IDE function to complete the class skeleton at the cursor. Right click on the editor and select Complete Class at Cursor

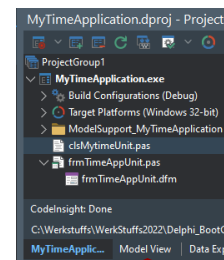


See:

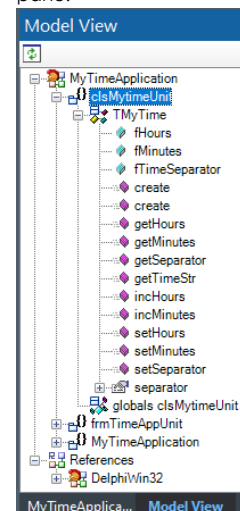
https://docwiki.embarcadero.com/RADStudio/Sydney/en/Using_Class_Completion



The Delphi IDE allows a developer to create a UML type class diagram using the Model view functionality.



Selecting and enabling the Model view option will create the following as part of the pane.



```

procedure TMyTime.incMinutes;
begin
  if (fMinutes = 59) then
    begin
      fMinutes := 0;
      Inc(fHours);
    end
  else inc(fMinutes);
end;

```

Auxiliary method to add 1 minute to the fMinutes attribute with some validation and logic applied to add 1 hour if the minutes pass 59

```

function TMyTime.getTimeStr:String;
var
  sTemp : String;
begin
  sTemp := '0';
  if fHours < 10 then
    sTemp := sTemp + IntToStr(fHours)
  else
    sTemp := IntToStr(fHours);

  sTemp := sTemp + ':';

  if fMinutes < 10 then
    sTemp := sTemp + '0' + IntToStr(fMinutes)
  else
    sTemp := sTemp + IntToStr(fMinutes);

  Result := sTemp;
end;

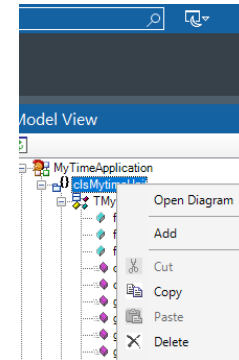
function TMyTime.getSeparator:char;
begin
  Result := fTimeSeparator
end;

procedure TMyTime.setSeparator(TS : char);
begin
  fTimeSeparator := TS;
end;
end.

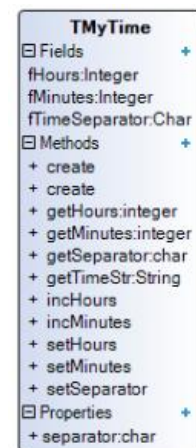
```

Auxiliary method to return the time in the format HH:MM where HH represent a double digit hour value the : the single character separator and the MM as the minutes presented with two digits

A class diagram can be created by the IDE. The developer has the option to right click on the class name and either Open the diagram or export the diagram to an image



The diagram created for our class will then render as follows



1.1.6 Implementing the class as part of an application.

At this stage the definition code for the class and the implementation code for the class have been completed.

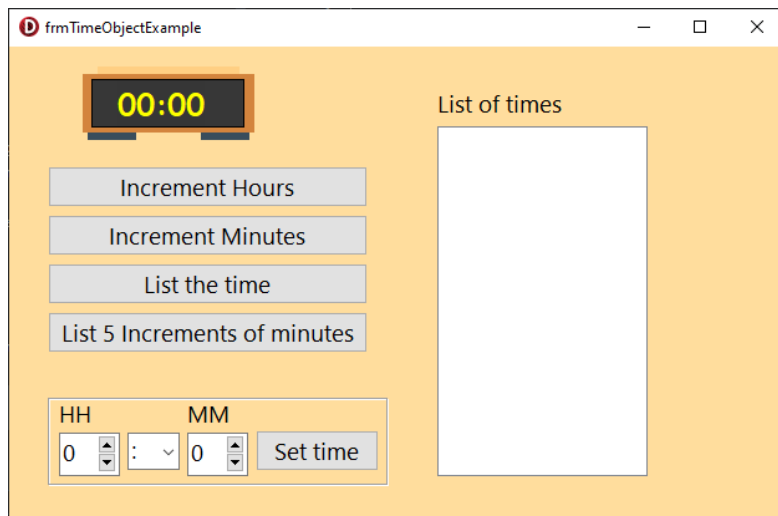
 Save the unit.

We can now use the class as part of our application.

The first thing to do is to include the class unit as part of the uses clause.

```
unit frmTimeAppUnit;  
  
interface  
  
uses  
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants, System.Classes, Vcl.Graphics,  
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs, clsMyTimeUnit;
```

Study the following user interface with buttons, a label, an image and a listbox as follows.



1.1.7 Discussion pertaining to the TMyTime implementation application

The complete form class code of the application is given below.

```
unit frmTimeAppUnit;  
  
interface  
  
uses  
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,  
  System.Classes, Vcl.Graphics,  
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls,  
  Vcl.Samples.Spin, Vcl.Imaging.pngimage, Vcl.ExtCtrls, clsMyTimeUnit;  
  
type  
  TfrmTimeObjectExample = class(TForm)  
    lblTime: TLabel;  
    lstTimes: TListBox;  
    Label1: TLabel;  
    btnIncHours: TButton;
```

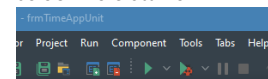
We need to include the name of our unit as part of the uses clause



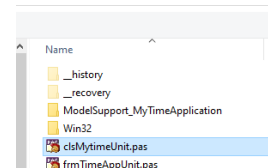
GOOD PRACTICE

It is a good practice to also include the class file to the project.

Project -> Add to Project
Select the class file



OS (C:) > Werkstuffs > WerkStuffs2022 > Delphi_Bo




```

btnIncMins: TButton;
btnListTime: TButton;
btnList5: TButton;
sedHours: TSpinEdit;
cmbSeparator: TComboBox;
sedMins: TSpinEdit;
btnSetTime: TButton;
Label2: TLabel;
Label3: TLabel;
Bevel1: TBevel;
Image1: TImage;
Shape1: TShape;
procedure FormCreate(Sender: TObject);
procedure FormShow(Sender: TObject);
procedure btnIncHoursClick(Sender: TObject);
procedure btnIncMinsClick(Sender: TObject);
procedure btnListTimeClick(Sender: TObject);
procedure btnList5Click(Sender: TObject);
procedure btnSetTimeClick(Sender: TObject);
private
{ Private declarations }

procedure updateTimeLabel;
public
{ Public declarations }
end;

```

```

var
frmTimeObjectExample: TfrmTimeObjectExample;

```

```
myTimeObj : TMyTime;
```

Declaring a TMyTime object
named **myTimeObj**.

```
implementation
```

```
{ $R *.dfm }
```

```

procedure TfrmTimeObjectExample.updateTimeLabel;
begin
  lblTime.Caption := myTimeObj.getTimeStr;
end;

```

Invoking / Calling the getTimeStr method to return the time as a
string in the format **HH sep MM**

```

procedure TfrmTimeObjectExample.btnIncHoursClick(Sender: TObject);
begin
  myTimeObj.incHours; // Calling the incHours method
  Self.updateTimeLabel; // Calling the user defined update label
                        // form class function
end;

```

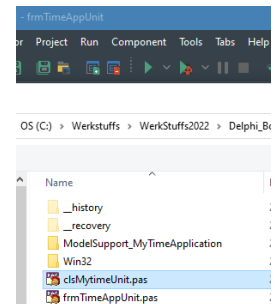
The self keyword is used to reference the current object in scope. This means that it
could be used to refer to the **frmTimeObjectExample** form. We could have replaced
Self with **frmTimeObjectExample**



Declaring objects

It is a good practice to also
include the class file to the
project.

Project -> Add to Project
Select the class file



Self

The word Self has
reference to
"MySelf" which
includes all
attributes and
methods of me.
Thus, stating Self.Speak is
referring the current object in
scope i.e., me. The VB.Net
language uses the keyword Me,
whilst Delphi uses Self.



Within the implementation of a
method, the identifier Self
references the object in which
the method is called. Self is
useful for a variety of reasons.
For example, a member
identifier declared in a class
type might be redeclared in the
block of one of the class'
methods. In this case, you can
access the original member
identifier as Self.Identifier.
([https://docwiki.embarcadero.com/RADStudio/Alexandria/en/Methods_\(Delphi\)](https://docwiki.embarcadero.com/RADStudio/Alexandria/en/Methods_(Delphi)))

```

procedure TfrmTimeObjectExample.btnIncMinsClick(Sender: TObject);
begin
    myTimeObj.incMinutes;
    Self.updateTimeLabel;
end;

```

```

procedure TfrmTimeObjectExample.btnList5Click(Sender: TObject);
begin
    var
        I : integer;
    for I := 1 to 5 do
        begin
            myTimeObj.incMinutes;
            lstTimes.Items.Add(myTimeObj.getTimeStr);
        end;
end;

```

```

procedure TfrmTimeObjectExample.btnListTimeClick(Sender: TObject);
begin
    lstTimes.Items.Add(myTimeObj.getTimeStr);
end;

```

```

procedure TfrmTimeObjectExample.btnSetTimeClick(Sender: TObject);
begin
    myTimeObj.setHours(sedHours.Value);
    myTimeObj.setMinutes(sedMins.Value);

```

Calling the mutator methods to pass the values from the spineditboxes as parameters to set the fHours and fMunites accordingly

```
myTimeObj.separator := cmbSeparator.Text[1];
```

Assigning the first character of the text selected in the combobox to the separator property of the class.

```

Self.updateTimeLabel;
end;

```

```

procedure TfrmTimeObjectExample.FormCreate(Sender: TObject);
begin
    myTimeObj := TMyTime.create;
end;

```

In this line of code, the **myTimeObj** object is instantiated. If we study the code from right to left (as the associativity of the **:=** operator is from right to left the following takes place. The create default constructor method is called, the object is created in memory and the address of where the object is created is assigned to the **myTimeObj** variable. Thus, referring to or using **myTimeObj** will refer to the correct position in the memory of where the object is created.

```

procedure TfrmTimeObjectExample.FormShow(Sender: TObject);
begin
    Self.updateTimeLabel;
    lstTimes.Clear;
end;
end.

```



Instantiation

When an object is made it is said that the object is instantiated. We have learned that an object is an instance of a class, so making the object is instantiating it.



Sometimes novice and vided programmers forget to instantiate an object for a user defined or built-in class.

This will result in a runtime error

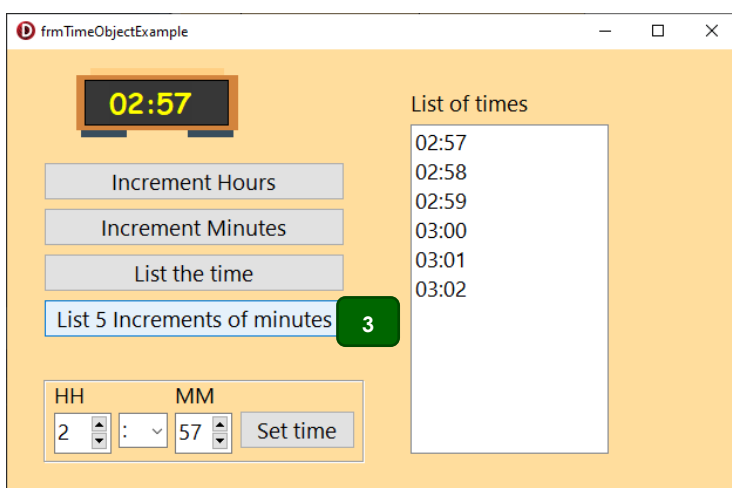
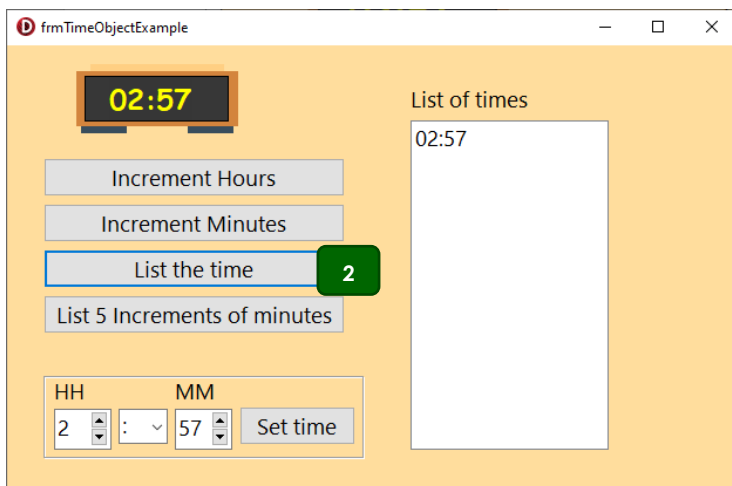
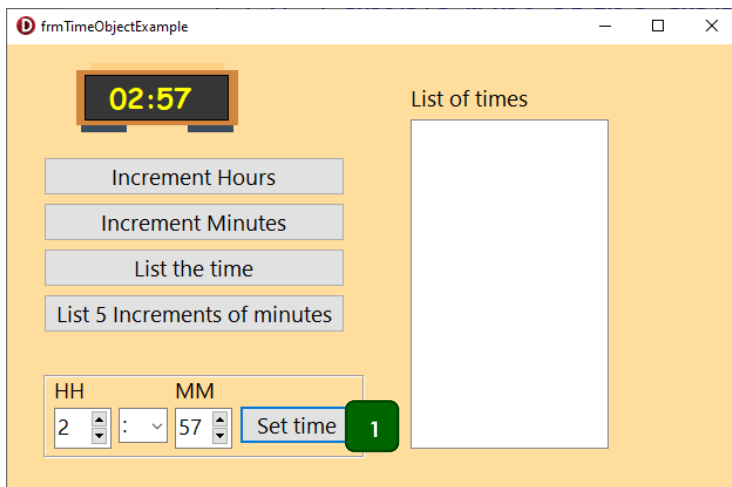


Which tells us that it could not find the corresponding object in memory as it has not been created yet.


So, when you get a similar error make sure that all your user defined objects are instantiated. ☺

1.1.8 Sample runs of the application

Some consecutive execution sample runs







DEITEL, H.M., DEITEL, P.J. 2001. C++ How to program. Third Edition. USA: Prentice Hall.

ERASMUS, H.G., FOURIE, M., PRETORIUS, C.M., 2002. Basic Programming Principles. SA:Heinemann.

FARRELL, J. 2001. Object Oriented Programming Using C++. Second Edition. Canada: Thompson Learning

LEE, R.C., TEPFENHART, W.M., 2001. UML and C++: A Practical Guide to Object-Oriented Development. 2nd Edition. USA:Prentice-Hall